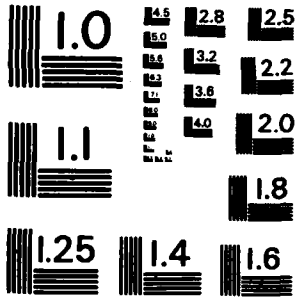
AD-A129 404 ADA/APSE: A SUCCESSOR TO CORAL 66 IN THE 80'S(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) T A WHITE JAN 83 RSRE-MEMO-3540 DRIC-BR-87073 1/1

UNCLASSIFIED F/G 9/2 NL

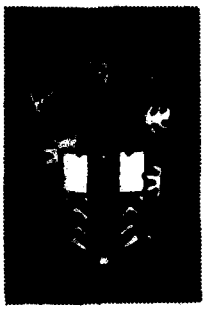


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

UNLIMITED

BR87073

(3)



RSRE
MEMORANDUM No. 3540

ROYAL SIGNALS & RADAR ESTABLISHMENT

AD A129404

Ada/APSE: A SUCCESSOR TO CORAL 66 IN THE 80's?

Author: T A D White

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

88 06 13 048
04 20 175

UNLIMITED

RSRE MEMORANDUM No. 3540

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3540

TITLE: Ada/APSE: A SUCCESSOR TO CORAL 66 IN THE 80's?

AUTHOR: T A D White

DATE: January 1983

SUMMARY

This paper briefly describes the historical context of, and the features offered by, the MOD standard real-time language, CORAL 66, and the recently developed US DoD standard language for embedded systems, Ada. It also sketches the principles behind the development of the APSE.

Ada/APSE will almost certainly replace CORAL 66 as the UK MOD preferred language in the mid- to late-80s.

Mention is made of the MOD preferred software design method MASCOT, and its role after a transition to Ada/APSE is explained.

Contents

1 Introduction

2 CORAL66

2.1 History

2.2 Features

2.2.1 Types and Expressions
2.2.2 Block Structure
2.2.3 Procedures
2.2.4 Independent Compilation
2.2.5 Control Structure
2.2.6 Access to Computer Hardware

3 Ada and APSE

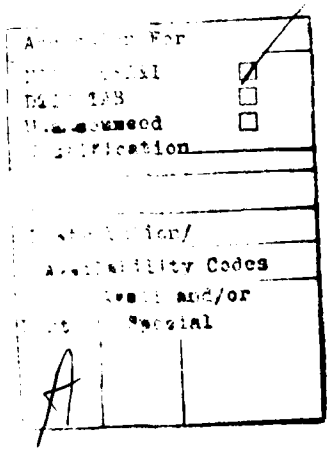
3.1 History

3.2 Features

3.2.1 Types
3.2.2 Packages
3.2.3 Separate Compilation
3.2.4 Generic Program Units
3.2.5 Exceptions
3.2.6 Tasks
3.2.7 Machine Representations and Foreign Languages
3.2.8 APSE

4 MASCOT

5 Concluding Remarks



1 Introduction

CORAL66 [1, 2] and Ada [3] are computer languages designed to address the same problem area. However, the sixteen years which separate their designs has left them with little else in common.

In the late fifties and early sixties limitations of slow and expensive hardware meant that great importance was placed on the maximum utilisation of computers' components. Programs were generally small and could be understood by one or two programmers who were themselves inexpensive to hire - even though they were specialists in what quickly became regarded as an "art". CORAL66 (Computer On-line Real-time Applications Language), an advanced language in its time, was designed to encourage the use of high-level languages during a period when intimate knowledge of a computer was required to achieve an efficient program.

The late seventies and early eighties have seen the introduction of chips which are cheap and disposable, and so fast and capacious as to be incomprehensible to the layman. Meanwhile, computers have been embedded in a greater variety of applications, including continuous processes, which are required to give reliable service: industrial plant, communications networks and civilian air traffic control are examples; more extreme examples are life-critical systems such as nuclear power stations and weapons systems where failure of the embedded software could result in accidental loss of life. Such large systems are rarely completely understood by one person; indeed, they are generally developed by teams of software engineers who work independently of one another and who are expensive to hire. It is now recognised that the production of a computer system is a task which must be brought under the management, quality control, and technical disciplines already used in other branches of engineering. Ada, or more strictly, the Ada Programming Support Environment (APSE) [4], has been designed to fulfil these goals.

2 CORAL66

2.1 History

The primary initiative behind the development of CORAL66 was the UK Ministry of Defence (MoD) requirement to introduce a standard high-level language into those defence applications which at the time were being implemented almost without exception in assembly language. The new language was to allow the programmer direct access to the hardware of the computer in order to make it attractive to assembler programmers and therefore a serious alternative. However, the language was to encourage the use of advanced programming language constructs which would enable better programs to be written more easily, with greater clarity and with improved integrity. Arguments levelled against the adoption of high-level languages also had to be countered: these were firstly that high-level languages resulted in larger programs which therefore ran more slowly, and secondly that the very need for a high-level language compiler implied the need for a large host computer on which programs could be developed. Design aims of CORAL66 therefore included a requirement that compilers produced efficient object code and were themselves small enough to be efficient when run on a modest mini-computer. The features of the language owe much to the then recently published Algol60 language which itself never received more than academic interest.

2.2 Features

The features of CORAL66 are informally described to show how data may be represented, how it may be protected, and how a program may be split into modules; in addition, the control structure is illustrated; finally, it is shown how the programmer is given access to the underlying machine to construct more complex data structures and interface to the target operating system.

Obviously a paper of this short length cannot describe the features of the language in detail; CORAL66 is fully specified in British Standard BS5905 [2].

Examples of CORAL66 source text follow the typographical convention of the British Standard.

2.2.1 Types and Expressions

Data objects in CORAL66 belong to one of only three types: INTEGER, FIXED or FLOATING.

It was the belief that the proper realm of computers was that of numbers; the programmer was required to exercise his own mind in order to express his ideas in digits. CORAL66 thus provides types to represent (as closely as possible in finite word length) the mathematical integer and real numbers. The real numbers are of two types: floating point numbers and fixed point numbers - the latter required because of the expense and poor performance of contemporary floating point hardware.

Thus, the CORAL66 programmer can declare numerical data objects:

```
INTEGER quantity ;

FLOATING height, range ;

FIXED(16,8) amps, volts ;
```

The usual arithmetic operators (+, -, *, / and parentheses) are provided to allow the construction of expressions.

For example, a programmer may write expressions:

```
quantity + 1

2 * pi * range

volts / amps

( a - b ) * ( a + b )
```

to achieve the obvious effects.

Arrays, based on these primitive types, are also available to the CORAL66 programmer.

2.2.2 Block Structure

CORAL66 retained an important concept popularised in Algol60 whereby programming language statements are bracketed together to form a block, and more importantly, where objects declared within such blocks are given restricted scope. The concept of blocks is retained in all modern algorithmic languages of good design.

Thus, a CORAL66 programmer can write a program fragment:

```
BEGIN
  FLOATING volts, amps ;
  .
  .
  .
END example block;
```

without concern that, by declaring new objects, he may be inadvertantly misusing other objects of the same name.

Local scope is also an early recognition of the importance of information hiding which allows the programmer to deny others access to, or even knowledge of, his own local objects.

2.2.3 Procedures

The concept of blocks and locally declared objects is used to powerful effect in procedures. Procedures are named fragments of program text which are written once but may be used many times. They may be provided with data from the main program and may return results to it.

For example, let us assume that the calculation of sines is required. The sine function can be embodied in a (typed) procedure, so:

```
FLOATING PROCEDURE sine ( VALUE FLOATING argument ) ;
BEGIN
  FLOATING result ;

  ... here would appear the source
  ... code for the sine function

  ANSWER result
END sine ;
```

Procedures were orginally introduced to allow code to be invoked from several places within a program and thus save precious space. This ability, with that of hiding local data objects or other unnecessary detail from the procedure's user, was the beginning of ideas to allow software fragments to be reusable - it can be seen that more sophisticated algorithms can be built up from simpler reusable fragments in what has been termed bottom-up fashion.

Furthermore, with a top-down view of program development, procedures allow a problem to be divided into smaller sub-problems each of which can be solved independently of the others.

2.2.4 Independent Compilation

Independent compilation recognises that there may be data and procedures which can sensibly form a collection to be used by programmers who need know nothing of the detailed program text. Independent compilation also recognises that programs may be too large to manage in a single compilation.

For example, assume a programmer knows nothing of the algorithmic details of a 'sine' procedure provided in a software library. He may nonetheless use it. For example, he may be able to write:

```
CORAL example program
LIBRARY trig functions
(
  FLOATING PROCEDURE sine ( VALUE FLOATING )
)
SEGMENT my program
BEGIN
  ... here would appear some use of the
  ... library 'sine' function, for example
  ... the assignment of an expression:
  ... height := range * sine ( elevation ) ;
END my program
FINISH
```

While providing obvious advantages, the independent compilation mechanism is severely limited by its ability to perform only name

checks on objects declared in other segments.

2.2.5 Control Structure

CORAL66 provides the classical Algol60-derived control structure for the assignment of values, the call of procedures, tests, loops and the transfer of control (jumps).

Thus, the CORAL66 programmer may assign values:

```
discriminant := sqrt ( b * b - 4 * a * c ) ;
```

he may test values:

```
IF height = 100 AND weight = 50 THEN ...
```

he may iterate a known number of times or until a condition is satisfied:

```
FOR i := 1 STEP 5 UNTIL 100 DO ...

FOR i := i + 1 WHILE i >= maximum DO ...
```

and finally he may transfer control to another named position in the program text:

```
GOTO end ;
```

2.2.6 Access to the Computer Hardware

The design of CORAL66 had to accommodate the construction of more sophisticated data objects; it also needed to allow the programmer to interact with real-time facilities provided by his machine. The solution adopted by CORAL66 to these problems, for which at the time no widely accepted solutions existed, was to allow the programmer easy direct access to the computer hardware. The aims of CORAL66 were thereby fulfilled, namely the provision of high-level language constructs where appropriate, but the availability of efficient machine level features where inefficiency may otherwise result. By this method it was hoped to attract programmers away from the exclusive use of assembly codes.

Thus, a CORAL66 programmer may specify, and can discover, the exact computer location of his data objects; he can then manipulate the objects directly - but only with reduced compiler checking. He is, in addition, allowed to write in machine-level code.

For example, a programmer may declare an object at a specified location:

```
ABSOLUTE ( INTEGER peripheral out / 10 ) ;
```

he may discover the location of a data object and manipulate it:

```
x := LOCATION ( height ) ;
```

```
[ x ] := [ x ] + 1 ; ( this will have altered 'height' )
```

with the effect that he may develop more sophisticated data structures:

```
last of list [ current ] := LOCATION ( next of list [ last ] ) ;
data of list [ current ] := result ;
next of list [ current ] := LOCATION ( last of list [ next ] ) ;
```

He may also descend into code, when machine facilities will be available:

```
COMMENT this code is a nonsense example;
CODE BEGIN
  SETA .0.1 ;
  ADDA .0.1 ;
  JNPA .SELF - 1
END code example;
```

3 Ada and APSE

3.1 History

The initiative for the development of the language Ada and its Programming Support Environment (APSE) came from the US Department of Defense (DoD) who wished to reduce their expenditure on software for embedded computer systems. The DoD had realised that much unnecessary expense was due to the large number of computer languages they supported. They wished, therefore, to follow the UK MoD example by introducing a single standard language throughout the US services, and by so doing to unify the language support that would be required.

Users' requirements for a standard language were drawn up and refined in a series of requirements specifications, the last two of which were known as the Ironman [5] and the Steelman [6]. Evaluation of many existing languages against the requirements showed that no contemporary language satisfied the DoD needs; the design of a new language was therefore proposed.

Four candidate languages were designed; code-named Blue, Green, Red and Yellow they were evaluated against the requirements and against each other, with the result that the Green and the Red [7] languages were taken on to a further stage of development. A second evaluation resulted in the Green language of CII-Honeywell-Bull being chosen to be the proposed DoD standard.

The language was named Ada after Augusta Ada Byron, Countess of Lovelace, who, as assistant to the nineteenth century mathematician Charles Babbage, became the world's first programmer.

It is the DoD's declared intention to protect the Ada programming language standard from misuse; to this end they have established a trademark and will require that compilers wishing to be known as "Ada" compilers shall have undergone testing and validation by the DoD or an agent of the DoD.

However, a new language is not enough. It had become apparent to those experienced with the specification, design, development, management and maintenance of large computer systems that a computer language by itself is not adequate. Additional goals of formal system requirements specification, program design, provability, software quality and maintainability are now as important as a technically competent language. It is recognised that a language needs software tools to support it. The ideas of support tools gave rise to the idea of the Ada Programming Support Environment (APSE).

The development of the APSE followed similar lines to the development of Ada itself; users' requirements were refined in a series f documents which culminated in the publication of the Ston . [4]. Stoneman identified several "layers" in a support nv nment: the KAPSE (the Kernal APSE) which would contain the machine dependencies; the MAPSE (the Minimal APSE) which would interface to a host operating system through the KAPSE and would

be that minimum set of tools which would allow simple text preparation, compiling and linking of an Ada program; and the APSE, which also interfaces through the KAPSE and is an open-ended set of tools to support the whole life-cycle of a software system.

It is not believed yet possible to reach international agreement on the details of a support environment so that one standard environment satisfying the Stoneman requirements seems unlikely. Currently (December 1982) there are four known efforts to realise the Stoneman requirements for an APSE: in the US the Air Force are procuring their Ada Integrated Environment (AIE) and the Army their Ada Language System (ALS); the UK MoD hopes to procure the MCHAPSE [8] - a Minimal APSE, but with an additional capability to support the telecommunications language CHILL [9]; there is also an initiative in Europe.

It is perhaps interesting to note that Ada and APSE represent a triumph of international co-operation. The US DoD adopted a policy of open development for the Ada requirement specification, the Ada language itself, and the requirements for the APSE. Thus each stage of development was open to, and received, comment and criticism world-wide. Since the adoption of the Green language as Ada further development has taken place and this too has been an exercise in international co-operation.

Furthermore, the significant European contribution to the development of Ada and the requirements specificiation for the APSE should not go unrecorded here - indeed, Ada was designed by a team led by the Frenchman Jean Ichbiah and Stoneman was written in the major part by Professor J Buxton of the University of Warwick.

3.2 Features

The features of Ada are informally described to show, in comparison to CORAL66, how data may be represented, how it may be protected, and how a program may be split into modules. Some consideration of the Ada tasking model and access to the underlying machine is also given.

Obviously a paper of this short length cannot describe the features of Ada in sufficient detail to do the language justice. Neither can it fully describe the APSE. Ada is defined in the Ada Language Reference Manual (LRM) [3], the APSE in the Stoneman [4].

Examples of Ada source text follow the typographical convention of the LRM, with the exception of the omission of bold typeface. One lexical convention, which may be strange to the CORAL66 programmer, is the use of an underscore character (_) in multi-word identifiers to represent what would otherwise have been a space, thus: ELAPSED_TIME.

3.2.1 Types

Computers have left the realm of numbers and are given the

ability to manipulate all kinds of objects which have types specified by the programmer. The programmer now no longer has to do his mental exercises to express his ideas in numbers, he can define types to reflect the properties of the objects he wishes to manipulate. In addition, the compiler is given the opportunity to carry out significantly more checks.

Thus, if an Ada programmer wished to describe the colour of a car he may write an enumeration type to specify the colours available:

```
type COLOUR is ( BLUE, BROWN, RED, WHITE, METALLIC_GREEN ) ;

COLOUR_OF_CAR : COLOUR ; -- the colour of a car is a colour

COLOUR_OF_CAR := WHITE ;
```

Ada types are strictly controlled. The Ada programmer needs to be more precise in his usage but receives the advantages of a readable program text which is checked by the Ada compiler.

For example, if having defined the type COLOUR, an Ada programmer defines objects of type 'SKILL':

```
type SKILL is ( WHITE, YELLOW, GREEN, BLUE, BLACK ) ; -- eg judo

SKILL_OF_COMPETITOR . SKILL ;
```

he would need to qualify use of the value 'WHITE' which appears twice. Thus:

```
COLOUR_OF_CAR := COLOUR ( WHITE ) ;
```

but

```
SKILL_OF_COMPETITOR := SKILL ( WHITE ) ;
```

The Ada programmer can define operations on his types as illustrated in the package DIRECTIONS.

Ada predefines the usual numeric types, a two-valued boolean type, and a character type to reflect the ASCII character set (ASCII is similar to the ISO-7-UK character set). Furthermore, it provides facilities to construct array types, record and pointer types. For example:

```
type SCHEDULE is array ( DAYS_OF_WEEK ) of BOOLEAN ;

type DATE is
   record
      DAY   : INTEGER range 1 .. 31 ;
      MONTH : NAME_OF_MONTH ;
      YEAR  : INTEGER range 0 .. 4000 ;
   end record ;
```

```
type LINK is access LIST_CELL ;
```

A subtype facility is provided which allows the programmer to constrain the values of a previously declared type available to an object.

For example:

```
type DAYS_OF_WEEK is ( MONDAY, TUESDAY,  WEDNESDAY, THURSDAY,
                       FRIDAY, SATURDAY, SUNDAY ) ;

subtype DAYS_OF_WEEKEND is DAYS_OF_WEEK range SATURDAY .. SUNDAY;
```

Additionally, derived types may be created where a new distinct type is modelled on an existing type. For example:

```
type VOLTS is new FLOATING 0.0 .. 240.0 ;

type AMPS  is new FLOATING 0.0 .. 13.0 ;
```

Notice that derived types recognise the true logical separation of a type, and its values, from another type, and its values (here we have volts and amps), even though the values may be based on the same underlying type (here the floating numbers). One is tempted to remember first-form introduction to physics equations when, on completion of a solution, the master would amend an apparently correct answer by adding the appropriate units eg x is not 3 but 3 volts!

The ability to clearly separate these logical properties carries a price, of course. The programmer may need to inform the compiler explicitly that, for example, an operator '/' operates not only on numbers, but also between values of type 'volt' and values of type 'amp' (producing one assumes a value of type 'ohm'). This information need only be presented to the compiler once, thereafter the programmer reaps the benefits of much improved checking.

3.2.2 Packages

Ada extends the ideas of blocks and procedures by providing facilities to package type definitions, operators and procedures into logical collections. The Ada programmer is given great control over the objects in his packages and will only release those details which are necessary to his users. Moreover, there is proper appreciation of the differences between the specification of a package (ie what it does) and its implementation (ie how it does it). Ada separates the two concepts so that a programmer using a package sees its specification (and may guess how it works), but only the package author sees the package body (ie implementation) and knows precisely how the package achieves its goal. The author may of course alter his method, but not the specification, without affecting his users. Should, however, the package author intend to alter his package specification the Ada compiler will be able

to report back the identity of those dependent packages which have been affected by the alteration.

The example is a package for a type DIRECTION and some operations on the values a direction can take. A user of the package would see the package specification, given first, and will know how a direction behaves, but he would not see the package implementation details (the body):

```
package DIRECTIONS is

   type DIRECTION is ( NORTH, WEST, SOUTH, EAST, DONT_CARE ) ;
   function TURN_LEFT  ( SOME_DIRECTION : DIRECTION )
                                          return DIRECTION ;
   function TURN_RIGHT ( SOME_DIRECTION : DIRECTION )
                                          return DIRECTION ;
   function ABOUT_TURN ( SOME_DIRECTION : DIRECTION )
                                          return DIRECTION ;

   -- rest of specification

end DIRECTIONS ;


package body DIRECTIONS is

   function TURN_LEFT ( SOME_DIRECTION : DIRECTION )
                                          return DIRECTION is
   begin
      if SOME_DIRECTION = DONT_CARE then
         raise CANT_TURN ;
      elsif SOME_DIRECTION = EAST then
         return NORTH ;
      else

         -- rest of function TURN_LEFT

   end TURN_LEFT ;

   -- remainder of source text of package body

end DIRECTIONS ;
```

Thus, the ideas of information hiding and the separation of specification and implementation have been given formal language constructs.

3.2.3 Separate Compilation

Separate compilation is a vastly more powerful concept than the independent compilation facility offered by CORAL66. Whereas independent compilation will check no more than the availability of the object to be used, separate compilation requires that full checking take place as if the object had been declared in the same module as its intended use.

Moreover, the Ada language definition admits not only compilation

units but also separately compiled subunits. Thus, the full definition of certain Ada objects are not given immediately within the compilation unit which references them. They are specified as a stub and a defining subunit is supplied separately. Hierarchical program development is thereby encouraged.

Complete details of separate compilation are given in the LRM; the description reveals the freedom given to programmers (and the consequent responsibilities placed on them), how programs may exploit separate compilation during their development, and how packages etc. are available for use.

3.2.4 Generic Program Units

Generic program units are a natural extension of the concept of the procedure. Just as a procedure may be parameterised to receive different values of a given type so generic program text can be parameterised to receive type information. They can thus abstract the manipulation of objects of many types.

For example, the manipulation of lists is generally independent of the type of element to be included in the list. List operations can be abstracted by generic program text, as shown:

```
generic

   SIZE : NATURAL ;
   type ELEMENT is private ;

package ON_LISTS is

   procedure APPEND ( L : in out LIST ; E : in ELEMENT ) ;
   function  HEAD   ( L : in LIST ) return ELEMENT ;
   procedure TAIL   ( L : in out LIST ) ;

   OVERFLOW, UNDERFLOW : exception ;

end ON_LISTS ;
```

Once generic program text has been accepted by the compiler it is possible to instantiate copies at which time generic parameters are supplied to tailor the new instance to the programmer's requirements.

Since generic program text can be used to support many instances, it is obvious that, once the generic program text has been written and tested (or even formally proved), the instances themselves can be used with far greater confidence. Such a facility is yet another facet of reusable software.

3.2.5 Exceptions

It has long been fashionable to escape from an unexpected or exceptional condition by jumping in panic to some named portion of program text. The Ada exception introduces into the language firstly the idea that things may go wrong, and secondly provides

a mechanism to assist the programmer to write correct and understandable program text to deal with his exceptional condition.

It is perhaps worth noting that the exception is provided to assist the programmer in dealing with the unusual condition rather than the rare but predictable event.

The example shows the declaration, use and handling of a supposed failure in a process which is monitored by a meter:

```
exception : PROCESS_FAILURE;                    — a declaration

if READING > 75 then
begin
   -- we have a high, but feasible, reading:
   — attempt to overcome the imbalance
   .
   .
   .
end ;


.
.
.
if READING > 100 then
   -- the reading "cannot" happen
   — assume something is wrong
   raise PROCESS_FAILURE ;                      — raising an exception
.
.
.


exception
   when PROCESS_FAILURE =>                      — handling an exception
   -- carry out remedial action:
   — issue warning of failure
   .
   .
   .
```

Several exceptions are predefined by the language; for example, CONSTRAINT_ERROR and NUMERIC_ERROR are predefined exceptions to trap the attempt to assign an illegal value to an object and to trap the failure to deliver a numerical result within the desired accuracy, respectively.

3.2.6 Tasks

Ada has defined within it a method for realising entities whose execution proceeds in parallel. The parallel processes are termed 'tasks'.

There are many models for parallel processing, the Ada task model is just one; it carries advantages and some disadvantages. The

model is an asymmetric one where tasks have entries; an entry of a task can be called by other tasks. A task accepts a call on one of its entries by executing an accept statement. The acceptance of an entry call is known as a rendezvous and it is during a rendezvous that synchronisation and the passing of parameters takes place.

Tasks, like packages, display the ability to separate the specification (in this case, how another programmer would see the task) from the body (ie the sequential source text which performs the task's actions).

Tasks may be declared directly or as a example of a task type.

For example:

```
task PROTECTED_DATA is

   entry READ  ( N : in INDEX; E : out ELEMENT ) ;
   entry WRITE ( N : in INDEX; E : in  ELEMENT ) ;

end PROTECTED_DATA ;

task body PROTECTED_DATA is

   TABLE : array ( INDEX ) of ELEMENT ;

begin

   loop

      select

         accept READ ( N : in INDEX; E out ELEMENT ) do
            E := TABLE ( N ) ;
         end READ ;

      or

         accept WRITE ( N : in INDEX; E : in ELEMENT ) do
            TABLE ( N ) : = E ;
         end WRITE ;

      end select ;

   end loop ;

end PROTECTED_DATA ;
```

3.2.7 Machine Representation and Foreign Languages

Ada acknowledges that embedded programs, while developed on a host computer with generous facilities, may be executed on target machines which are poorly endowed with spare capacity. Furthermore, it is recognised that there is considerable investment in software written in languages other than Ada. To enable the programmer to optimise his program in terms of the

target machine and existing software modules machine representations, code and foreign language inserts are allowed. Thus, the programmer is allowed to specify the representation of this data, descend into code (but only in a strictly controlled fashion), and interface to foreign languages to import into his Ada program subprograms (procedures) written in a language other than Ada. Examples give the flavour of these low-level facilities:

```
for SKILL use                    — the type SKILL is defined in 3.2.1
   (WHITE => 1, YELLOW => 2, GREEN => 3, BLUE => 4, BLACK => 5);
```

Note that a 'use clause' such as that given does not imply that the abstraction of the type is degraded in any way; so that, for example, an expression 'YELLOW + 1' will have no meaning in spite of the observation that YELLOW is represented by the value 2 and GREEN, a possible meaning for the expression, is represented by the value 3.

An example to give the idea behind interfacing to a foreign language is given:

```
package CORAL66_TRIG_FUNCTIONS is

   function  SINE   ( ARGUMENT : FLOAT ) return FLOAT ;
   function COSINE  ( ARGUMENT : FLOAT ) return FLOAT ;

private

   pragma INTERFACE ( CORAL66,  SINE   ) ;
   pragma INTERFACE ( CORAL66, COSINE  ) ;

end CORAL66_TRIG_FUNCTIONS ;
```

The LRM contains exhaustive description of the representation specifications available.

3.2.8 APSE

As mentioned briefly it has been realised that no programming language, however good, is sufficient in itself. The Ada language is unlikely to be regarded as a success in the UK, in spite of its technical advantages, unless it can offer positive advances in other areas.

It is hoped that the APSE will provide these advances. The APSE philosophy will enable the programmer to work more effectively since within each similar APSE product there will be: firstly, an interface, standard to all similar APSEs, which is provided to hide details of a host operating system by ensuring that a defined set of primitive facilities are available - the interface and primitives are embodied in the Kernal APSE (KAPSE); secondly, a basic but useful set of competent tools, which exploit the facilities provided by the KAPSE - this set of tools is known as the MAPSE, and, in the UK, it is hoped that they will be provided by the MCHAPSE product; and thirdly, the ability to exploit the much improved standardisation of the Ada language and APSE

interfaces which should encourage the production of APSE tools which address all aspects of the software life-cycle, and furthermore allow the easy migration of such tools from installation to installation.

4 MASCOT

Within the UK defence community it has become practice to assist software development with the design method MASCOT [10]; indeed, reference is often made to the pair CORAL/MASCOT since the MASCOT ACP diagram, produced by the design process, is realised by constructing a network of modules specified in the programming language CORAL66. However, the use of CORAL66 is not inherent in MASCOT, which is language independent.

It would be incorrect to attempt to compare CORAL66/MASCOT with Ada or Ada/APSE ie a language and software design method with an unsupported language or with a language supported to the great extent expected of an APSE. The proper comparison it that chosen - the comparison of the programming language CORAL66 with the programming language Ada.

MASCOT may well be incorporated into the UK APSE since it is an established technique and, moreover, the advance desired of the introduction of Ada is that offered by the APSE, which, as a general support environment, should assist software design.

5 Concluding Remarks

This brief paper has discussed the features offered by the computer languages CORAL66 and Ada; it has further described the support offered to Ada by the Ada Programming Support Environment, the success of which is recognised to be of paramount importance since experience in the UK, with its present language standard, has shown that the production of a software system is an undertaking for which a programming language, by itself, is not enough.

References

[1] Woodward P, Wetherall P and Gorman B, Official Definition of CORAL66, HMSO, 1970.

[2] Specification for the Computer Programming Language CORAL66, BS5905, November 1980.

[3] Reference Manual for the Ada Programming Language, US DoD, July 1982.

[4] Requirements for Ada Programming Support Environments (Stoneman), US DoD, February 1980.

[5] Requirements for High Order Computer Programming Languages (Ironman), US DoD, January 1977.

[6] Requirements for High Order Computer Programming Languages (Steelman), US DoD, June 1978.

[7] Red Language Reference Manual, Intermetrics, March 1979.

[8] Requirement Specification for the Minimal Ada/CHILL Programming Support Environment - the MCHAPSE, RSRE, July 1982.

[9] CHILL Language Definition (Rec.Z.200), CCITT, March 1982.

[10] The Official Handbook of MASCOT, MSA, December 1980.

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..... Unclassified .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Memorandum 3540	3. Agency Reference	4. Report Security Classification Unclassified
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location Royal Signals and Radar Establishment		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		
7. Title Ada/APSE: A Successor to Coral 66 in the 80's?			
7a. Title in Foreign Language (in the case of translations)			
7b. Presented at (for conference papers) Title, place and date of conference			

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
White, T A D				

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement Unlimited

Descriptors (or keywords)

•

continue on separate piece of paper

Abstract This paper briefly describes the historical context of, and the features offered by, the MOD standard real-time language, CORAL 66, and the recently developed US DoD standard language for embedded systems, AdA. It also sketches the principles behind the development of the APSE.

Ada/APSE will almost certainly replace CORAL 66 as the UK MOD preferred language in the mid- to late-80s.

Mention is made of the MOD preferred software design method MASCOT, and its role after a transition to Ada/APSE is explained.

END

DATE
FILMED

7 83

DTIC